UNIVERSITY AT ALBANY
State University of New York

COLLEGE OF ENGINEERING AND APPLIED SCIENCES
DEPARTMENT OF COMPUTER SCIENCE
ICSI213/IECE213 Data Structures

Project 02 Created by Qi Wang

Click here for the project discussion recording.

Table of Contents

- **Part I: General Project Information**
  All projects are individual projects unless it is notified otherwise.
- All projects must be submitted via Blackboard. No late projects or e-mail submissions or hard copies will be accepted.
- Two submission attempts will be allowed on Blackboard. **Only the last attempt will be graded.**
- Work will be rejected with no credit if
  - The project is late.
  - The project is not submitted properly (wrong files, not in required format, etc.). For example,
    - The submitted file can't be opened.
    - The submitted work is empty or wrong work.
    - Other issues.
  - The project is a copy or partial copy of others' work (such as work from another person or the Internet).
- **Students must turn in their original work. Any cheating violation will be reported to the college. Students can help others by sharing ideas, but not by allowing others to copy their work.**
- Documents to be submitted as a zipped file:
  - **UML class diagram(s)** – created with Violet UML or StarUML
  - **Java source file(s) with Javadoc style inline comments**
  - **Supporting files if any** (For example, files containing all testing data.)
- Students are required to submit a design, all error-free source files with Javadoc style inline comments, and supporting files. Lack of any of the required items will result in a really low credit or no credit.
- Your TA will grade, and then post the feedback and the grade on Blackboard if you have submitted it properly and on time. If you have any questions regarding the feedback or the grade, please reach out to the TA first. You may also contact the instructor for this matter.

Part II: Project grading rubric

Components	**Max points**
UML Design (See an example in part II.)	Max. 10 points
Javadoc Inline comments (See an example in part II.)	Max. 10 points
The rest of the project	Max. 40 points

All projects will be evaluated based upon the following software development activities.

Analysis:

- Does the software meet the exact specification / customer requirements?
- Does the software solve the exact problem?

Design:

- Is the design efficient?

Code:

- Are there errors?
- Are code conventions followed?
- Does the software use the minimum computer resource (computer memory and processing time)?
- Is the software reusable?
- Are comments completely written in Javadoc format?
    a. Class comments must be included in Javadoc format before a class header.
    b. Method comments must be included in Javadoc format before a method header.
    c. More inline comments must be included in either single line format or block format inside each method body.
    d. All comments must be completed in correct format such as tags, indentation etc.

Debug/Testing:

- Are there bugs in the software?

Documentation:

- Complete all documentations that are required.

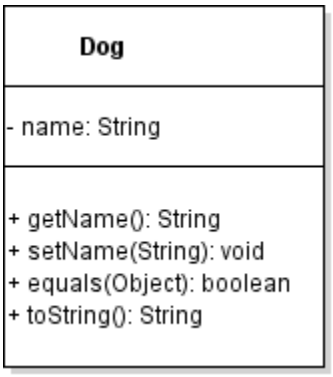
Part III: Examples on complete a project from start to finish

To complete a project, the following steps of a software development cycle should be followed. These steps are not pure linear but overlapped.

Analysis-design-code-test/debug-documentation.

1) Read project description to understand all specifications(**Analysis**).
2) Create a design (an algorithm for method or a UML class diagram for a class) (**Design**)
3) Create Java programs that are translations of the design. (**Code/Implementation**)
4) Test and debug, and (**test/debug)**
5) Complete all required documentation. (**Documentation**)

The following shows a sample design. The corresponding source codes with inline Javadoc comments are included on next page. How to test/debug a software is included on the following pages.



```
import java.util.Random;

/**
 * Representing a dog with a name.
 * @author Qi Wang
 * @version 1.0
 */
public class Dog{
    /**
     * The name of this dog
     */
    private String name;


    /**
     * Constructs a newly created Dog object that represents a dog with an empty name.
     */
    public Dog(){
        this("");
    }
    /**
     * Constructs a newly created Dog object wit
     * @param name The name of this dog
     */
    public Dog(String name){
        this.name = name;
    }

    /**
     * Returns the name of this dog.
     * @return The name of this dog
     */
    public String getName(){
        return this.name;
    }

    /**
     * Changes the name of this dog.
     * @param name The name of this dog
     */
    public void setName(String name){
        this.name = name;
    }

    /**
     * Returns a string representation of this dog. The returned string contains the type of
     * this dog and the name of this dog.
     * @return A string representation of this dog
     */
    public String toString(){
        return this.getClass().getSimpleName() + ": " + this.name;
    }


    /**
     * Indicates if this dog is "equal to" some other object. If the other object is a dog,
     * this dog is equal to the other dog if they have the same names. If the other object is
     * not a dog, this dog is not equal to the other object.
     * @param obj A reference to some other object
     * @return A boolean value specifying if this dog is equal to some other object
     */
    public boolean equals(Object obj){
        //The specific object isn't a dog.
        if(!(obj instanceof Dog)){
            return false;
```

Class comments must be written in Javadoc format before the class header. A **description** of the class, author information and version information are required.

Comments for fields are required.

open {

TAB

TAB

Method comments must be written in Javadoc format before the method header. the first word must be a **capitalized** verb in the **third** person. Use punctuation marks properly.

open {

TAB

TAB

A description of the method, comments on parameters if any, and comments on the return type if any are required.

A Javadoc comment for a formal parameter consists of three parts:

- parameter tag,
- a name of the formal parameter in the design ,
  (The name must be consistent in the comments and the header.)
- and a phrase explaining what this parameter specifies.

A Javadoc comment for return type consists of two parts:

- return tag,
- and a phrase explaining what this returned value specifies

More inline comments can be included in single line or block comments format in a method.

```
            }

            //The specific object is a dog.
            Dog other = (Dog)obj;
            return this.name.equalsIgnoreCase(other.name);
        }
}
```

Part IV:

A. How to test a software design?

There can be many classes in a software design.

1. **First, create a UML class diagram containing the designs of all classes and the class relationships (For example, is-a, dependency or aggregation).**
2. **Next, test each class separately.**
   Convert each class in the diagram into a Java program. When implementing each class, a driver is needed to test each method included in the class design. In the driver program,
   i. Use the constructors to create instances of the class(If a class is abstract, the members of the class will be tested in its subclasses.). For example, the following creates Dog objects.
   Create a default Dog object.
   `Dog firstDog = new Dog();`
   Create a Dog object with a specific name.
   `Die secondDog = new Dog("Sky");`

   ii. Use object references to invoke the instance methods. If an instance method is a value-returning method, call this method where the returned value can be used. For example, method getName can be called to return a copy of firstDog's name.

```
String firstDogName;
  …
firstDogName = firstDog.getName();
```

   You may print the value stored in firstDogName to verify.

   iii. If a method is a void method, invoke the method that simply performs a task. Use other method to verify the method had performed the task properly. For example, setName is a void method and changes the name of this dog. After this statement, the secondDog's name is changed to "Blue".
   `secondDog.setName("Blue");`

   getName can be used to verify that setName had performed the task.

   iv. Repeat until all methods are tested.

- **And then, test the entire design by creating a driver program and a helper class of the driver program.**
  i. Create a helper class. In the helper class, minimum three static methods should be included.

```
public class Helper{

    //method 1
    public static void start(){
        This void method is decomposed.
        It creates an empty list.
        It calls the create method to add a list of objects to the list.
        And then, it calls the display method to display the list of objects.
    }

    //method 2
    public static returnTypeOrVoid create(parameters if any) {
```

```
            This method creates a list of objects using data stored in text files.
        }

        //method 3
        public static returnTypeOrVoid display(parameters if any) {
            This method displays a list of objects.
        }
    }
```

ii. Create a driver program. In *main* of the driver program, call method *start* to start the entire testing process.

```
public class Driver{
    public static void main(String[] args){
        Helper.start();
    }
}
```

Notice that the driver and its helper class are for testing purpose only. They should not be included in the design diagram.

B. Project description

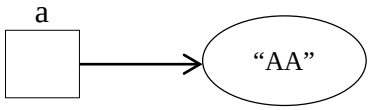
Project 2 ADT *LinkedString*

Java *String* class is composed of a collection of characters and a set of operations on the characters. In this project, you will create a reference-based implementation on some *String* operations such as *chartAt, concat, isEmpty, length, substring*. A doubly linked list must be used as the data structure. The class is called *LinkedString*. This class must be implemented so that objects of *LinkedString* are immutable as *String* objects.

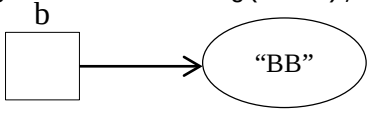
In Java, a s*tring* is an immutable object (its internal states cannot be changed once it's created). Immutable means that once the constructor has completed execution that instance made can't be altered. This is useful as it means you can pass references to the object around, without worrying that someone else is going to change its contents. Any method that is invoked which seems to modify the value, will actually create another String. For example, three *String* objects, a, b, and ab, are created in the following code segment.

```
String a = new String("AA");
String b = new String("BB");
String ab = a.concat(b);
```

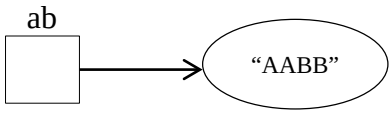
After `String a = new String("AA");` is executed, a new *String* object a is created.



After `String b = new String("BB");` is executed, another new *String* object `b` is created.



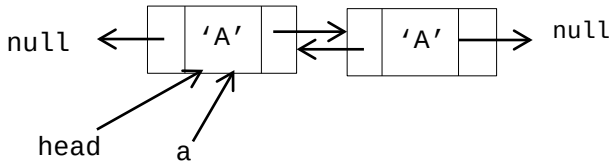
After `String ab = a.concat(b);` is executed, another new *String* object ab is created. *String* a(*this* string) and *String* b (a string passed into method `concat`) are not changed due to String immutability. Method `concat` simply copies the contents of `a` and `b`, and uses them to make a new *String* object.



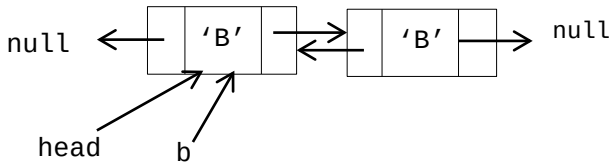
The *LinkedString* class must be implemented so that *LinekdString* objects are immutable. The *LinkedString* class must use a doubly linked list, a different data structure from the one used by the *String* class, to store a collection of characters. One character per node. This data structure is *LinkedString* 's internal state. An immutable *LinkedString* object means its linked list can't be altered once the object is created. All characteristics and behaviors of *LinkedString* class must be designed with the same logic as Java *String* class. When a *LinkedString* object calls a method, this *LinkedString* object and *LinkedString* object(s) passed into this method must be unchanged during execution of this method. If the method returns a *LinkedString* object, a new *LinkedString* object must be made. The following shows how object immutability can be enforced when implementing method `concat`. For example, three *LinkedString* objects, a, b, and ab, are created in the following code segment.

```
LinkedString a = new LinkedString ("AA");
LinkedString b = new LinkedString ("BB");
LinkedString ab = a.concat(b);
```

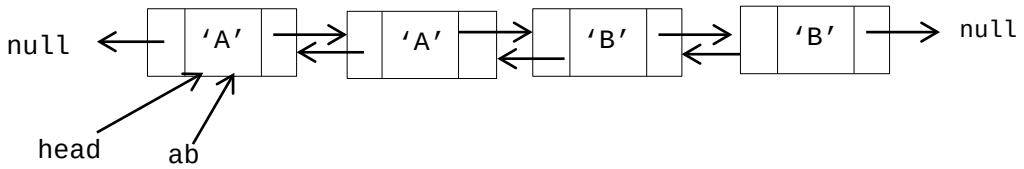
After `LinkedString a = new LinkedString("AA");` is executed, a new *LinkedString* object `a` is created with all characters stored in a doubly linked list. Each node contains a *Character* element, a successor and a predecessor.



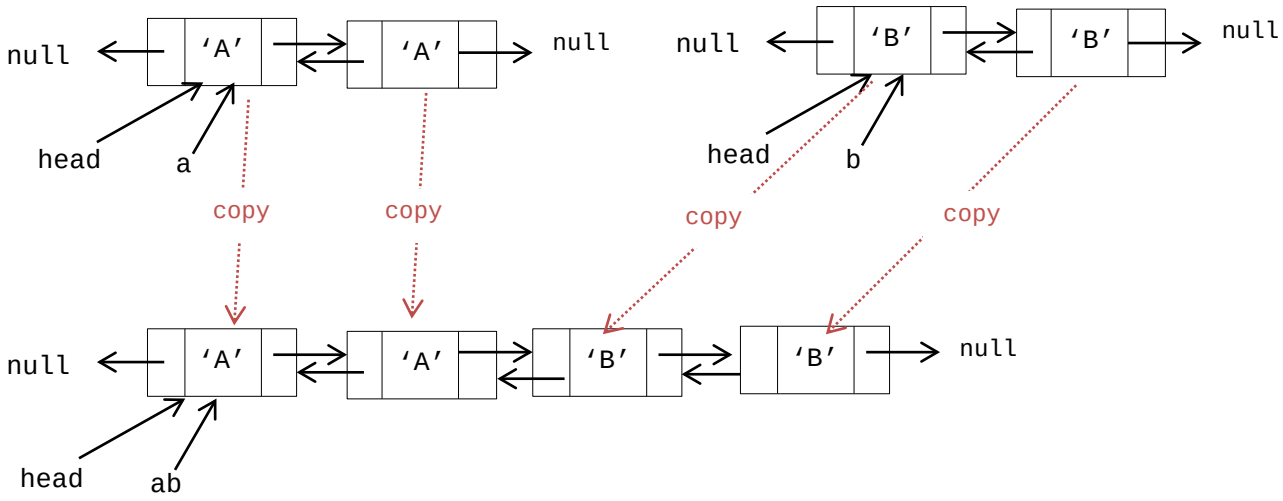
After `LinkedString b = new LinkedString("BB");` is executed, another new *LinkedString* object `b` is created with all characters stored in a doubly linked list. Each node contains a *Character* element, a successor and a predecessor.



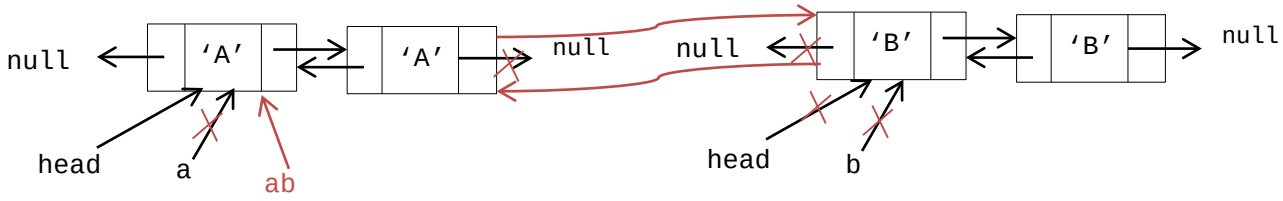
After `LinkedString ab = a.concat(b);` is executed, another new *LinkedString* object `ab` is created with all characters stored in a doubly linked list. Each node contains a *Character* element, a successor and a predecessor.



Method `concat` must be implemented in a way in which a new linked string is made without modifying this linked string `a` and the other linked string `b` to enforce object immutability. In order to do this, method `concat` should simply copy characters and use them to make a new linked string.



Modifying this linked string a or other linked string b like this would violate object immutability property.



Carefully implement each method, and make sure object immutability is maintained.

Specification/Analysis:

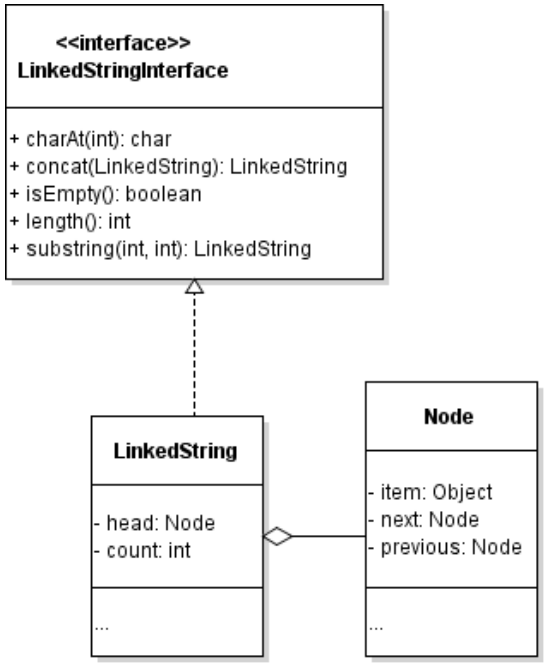
- **A Node:**
  The *Node* class can be instantiated to create a node containing an element of *Object* type, a successor and a predecessor. When using it in *LinkedString*, reference type *Character* can be used for the *element* of the node.

- ***LinkedString* Operations:**
  Note: All operations must be implemented to enforce object immutability. This means this linked string and other linked strings involved cannot be modified once after they are created.
  - ✓ return the *char* value at the specified index. (*char charAt(int)*).
  - ✓ concatenate a specified linked string to the end of this linked string (*LinkedString concat(LinkedString)*).
  - ✓ returns true if and only if the length of this linked string is 0. (*boolean isEmpty()*)
  - ✓ return the length of this linked string (*int length()*).
  - ✓ return a new linked string that is a substring of this linked string (*LinkedString substring(int, int)*).

Design:

Complete a UML diagram to include all classes. An interface class is usually defined to specify what the operations do. A class implementing this interface provides implementations to specify how the operations are implemented. Exceptions should be considered when operations are designed.

In the design, you should include the design of *LinkedString*, the design of a *Node* class that is needed for a doubly linked list, etc.. The following shows **part** of the design for this project.



Code/Implementation:

Implementation includes selecting a data structure, implementation of constructors and the operations.

A doubly linked list with an external reference to the head must be used, as the data structure. A doubly linked list is a reference to the head of a doubly linked list.

Three overloading constructors should be provided to make a linked string from an empty list, a *char* array or a *String* object.

- create an empty *LinkedString* instance.
  A new character linked list is allocated so that it represents the sequence of 0 characters currently contained in the character list argument. (*LinkedString()*).
- create a *LinkedString* instance containing a sequence of characters.
  A new character linked list is allocated so that it represents the sequence of characters currently contained in the character list argument. (*LinkedString(char[])*).
- create a *LinkedString* instance containing same sequence of characters as a *String* instance.
  A new character linked list is initialized so that it represents the same sequence of characters as the *String* argument (*LinkedString(String)*).

When implementing overloading methods/constructors, you should write all the codes in one method, the one that has most parameters, and let others invoke/reuse the method. In the case, the last two constructors all have one parameter. Because a *String* object can be converted into a *char* array, therefore, all the codes should be written in the second constructor. The first constructor and the third constructor should invoke the second constructor using *this* reference.

When implementing other methods, make sure all *LinkedString* objects involved are immutable. It may be helpful to create helper methods for some of the *LinkedString* methods. If so, those helper methods should be private methods. Javadoc comments should be included. Class comments must be included right above the corresponding class header.

Method comments must be included right above the corresponding method header. All comments must be written in Javadoc style.

Debug/Testing:

Note: It is required to store all testing data in a file. It is required to use decomposition design technique.

To test the *LinkedString* design, all operations must be tested. In general, a list of *LinkedString* object is created, and then, use the list to test other operations. It is not efficient to write everything in *main*. Method *main* should be small and the only method in a driver program. A helper class should be created to assist the driver. In the *Helper* class, minimum three static methods should be included. Method *start* is decomposed to create an array list of *LinkedString* objects(create). And then, use them in other method(*displayAndMore*). Method *create* should add the linked strings into the list. Method *displayAndMore* should use the list to test other methods. More methods can be added to test other operations. Method *main* should call *start* from the driver program to start the entire testing process.

Test the entire design by creating a driver program and a helper class of the driver program.

- Create a helper class. In the helper class, minimum three static methods should be included.

```
public class Helper{
    //method 1
    public static void start(){
        This void method is decomposed.
        It creates an empty array list that can be used to store a list of LinkedString objects.
        It calls create with a reference to the array list.
        create adds a list of LinkedString objects into the array list.
        It calls displayAndMore with a reference to the array list.
        displayAndMore uses the list to test other methods.
    }

    //method 2
    public static returnTypeOrVoid create(A reference to an array list){
        Using data stored in text files to make LinkedString objects.
        Add the LinkedString objects into the array list.
        Note: In this case, you will read the testing data, use some as strings and some as character
              arrays in order to test all constructors.
    }

    //method 3
    public static returnTypeOrVoid displayAndMore(A reference to an array list){
          For every two adjacent Linked strings in the list, if they are not empty(isEmpty returns false),
              Call length and print the returned length of each linked string.
              Call charAt and print the returned first character of each linked string.
              Call substring to get the first character and print the returned string of each linked string.
              Call concat to concatenate and print the length of each concatenated linked string.
    }
}
```

- Create a driver program. In *main* of the driver program, call method *start* to start the entire testing process.

```
public class Driver{
    public static void main(String[] args){
          Helper.start();
    }
}
```

The sample testing file may contain items like this:

```
Olivia
Oliver
Amelia
Harry
Isla
Jack
Emily
George
Ava
Noah
Lily
Charlie
Mia
Jacob
Sophia
Alfie
Isabella
Freddie
Grace
Oscar
…
```

Documentation:

Complete all other documents needed.